LAB REPORT

LAB 7

TASK 1

1.1) I downloaded mnist_784 directly from sklearn.datasets and read the mnist.data as file, printed shape of data and also the keys of complete dataset mnist, output for which is attached below.

```
Shape of DataSet is :  (70000, 784)
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names',
'DESCR', 'details', 'categories', 'url'])
```

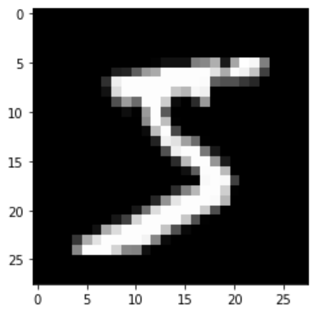
1.2) 'minst.data' only has colour values of each pixels in one column so for 28x28 pic we have 784 columns (features) and 70,000 total pictures and the corresponding target value is stored in mnist["target"], Also the description similar to this can be found out using ---> mnist["DESCR"]

```
2. print(mnist["DESCR"])
```

1.3 I reshaped the image to 28x28 matrix and then printed it using the Image from PIL, using

```
pic=mnist.data[0]
plt.imshow(pic.reshape(28,28),cmap='gray')
# Visualizing the first images in the dataset by reshaping the data (28
 x 28)
```

OUTPUT -------------->



1.4 (i) Using in-built function first I used standard scaler on data to fit-transform it to file_scaled, this was done to make sure that no feature gets advantage just because the average is very high, thereby enhancing our data, and then using PCA in-built function I reduced feature to only 2 features from 784 features. I also printed amount of Variance contributed by both the Principal Components.

```
scaler=StandardScaler()
file_scaled=scaler.fit_transform(file)
```

```
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(file_scaled)
df_PCA_n=pd.DataFrame(data=principalComponents)

print("Original shape:   ", file.shape)
print("Transformed shape:", df_PCA_n.shape)


print("Amount of Variance contributed by both Principal Components : ",
pca.explained_variance_ratio_)
```

OUTPUT :

```
Original shape:    (70000, 784)
Transformed shape: (70000, 2)
Amount of Variance contributed by both Principal Components :
[0.05642719 0.04041226]
```

1.4 (ii) Now we are fnding how many features we will have to retain if we want variance to be above 0.9, using the in-built functions.

```
pca = PCA(.9)        # for 0.9 variance to be retained
principalComponents = pca.fit_transform(file_scaled)
df_PCA_Var = pd.DataFrame(data = principalComponents)

print("Number of Principal Components required to have 0.9 variance : "
,pca.n_components_)
```

OUTPUT :

```
Number of Principal Components required to have 0.9 variance :  238
```

So atleast 238 features are to be retained, if we want the variance to be above 0.9.

TASK 2

2.1) First I split the dataset into 5000,65000 samples set, then I used the 5000 set and converted it into np.array for easier operations and in-built functions, then I shifted the origin to mean position of point cloud, and then used the below formula to calculate covariance matrix:

```
Original_file=pd.DataFrame(mnist.data)
print("Shape of Original DataSet (As imported) is : ",Original_file.sha
pe)

excess_data, data_final = train_test_split(Original_file, test_size=0.0
7142857142)

print("Data to Be Used has shape : ",data_final.shape)
```

```
matrix=np.array(data_final)     # Getting our DataFrame in np.array form
 for easier calculations
matrix_mean = np.mean(matrix, 0)

# Covariance Matrix Calculation

matrixShifted = matrix -
 np.mean(matrix, 0)                                  # Shifting the cetr
e to Mean Point of N-Dimensional Point Cloud
covMatrix = 1 / matrix.shape[0] * np.matmul(matrix.T, matrixShifted)
```

2.2) Defined a function called pca(), in which I used in built function linalg.eigh to find the eigenValues, eigenVectors, then sorted all in accordance to eigenValues, and then After multiplication of original matrix and all Eigenvectors we get reducedMatrix, and then I return all three for future reference. Then passed my matrix to this and then printed Top 5 eigenvalues.

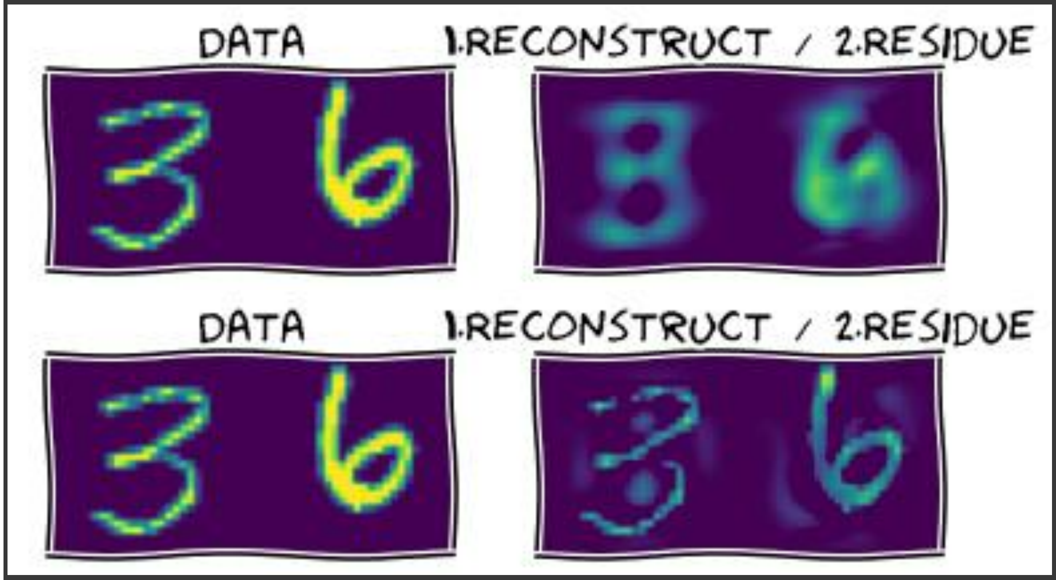
```
def pca(matrix):

  eigenValues, eigenVectors = np.linalg.eigh(covMatrix)
       # Finding Eigenvalues and Eigenvectors using in-built function


  # Sorting the EigenVectors on basis of descending EigenValues
  sortedIndices = np.argsort(eigenValues)[::-1]
  sortedEigenvectors = eigenVectors[:,sortedIndices]
  sortedEigenvalues = eigenValues[sortedIndices]


  # After multiplication of original matrix and all Eigenvectors we get
 reducedMatrix
  reducedMatrix=np.matmul(matrix, sortedEigenvectors)


  return reducedMatrix, sortedEigenvectors, sortedEigenvalues
reducedMatrix, sortedEigenvectors, sortedEigenvalues = pca(matrix)

for i in range(5):
  print(i+1,'Eigen Value : ',sortedEigenvalues[i])
```

OUTPUT :

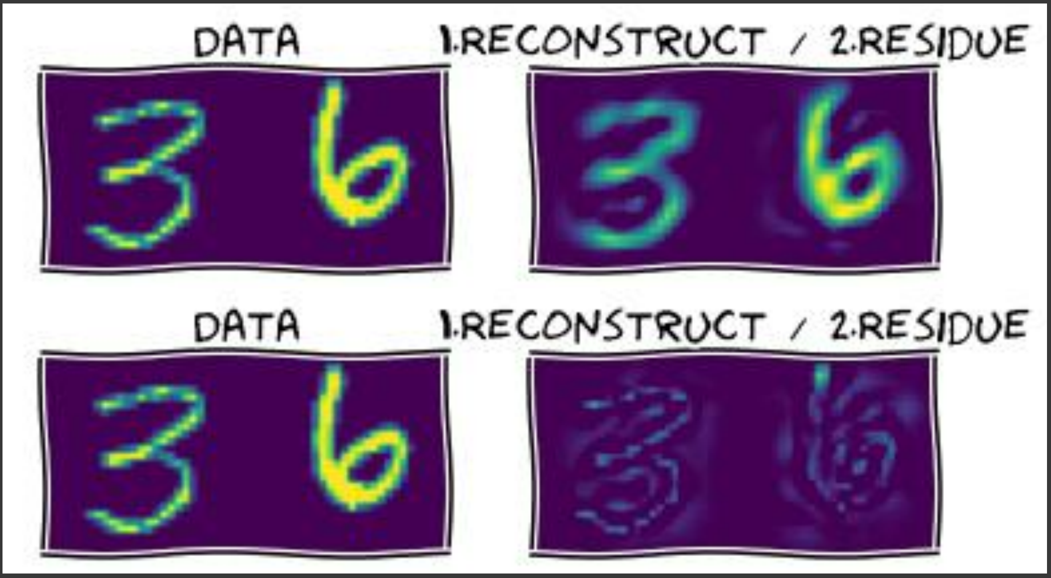
```
1 Eigen Value :  330342.4175685054
2 Eigen Value :  246836.55070709332
3 Eigen Value :  210662.25281229368
4 Eigen Value :  192600.77698496732
5 Eigen Value :  162620.39640857998
```

2.3,2.4,2.5) Now made a function to reconstruct the matrix back, and used the reverse transform by multiplying the reducedMatrix with sortedEigenvectors (till the K we requires as in the function), and also made the residue matrix. Then I made a function to plot the comparison between two matrices using plt.xkcd(), this will be used for both Reconstructed Matrix and the Residue Matrix (In one plot for a K we have first for reconstructed and then below it for residue). Now OUTPUT for different K (also I have given output for K=784, which should be exactly same as Original bcz all Principal Components have been used ----- was same)

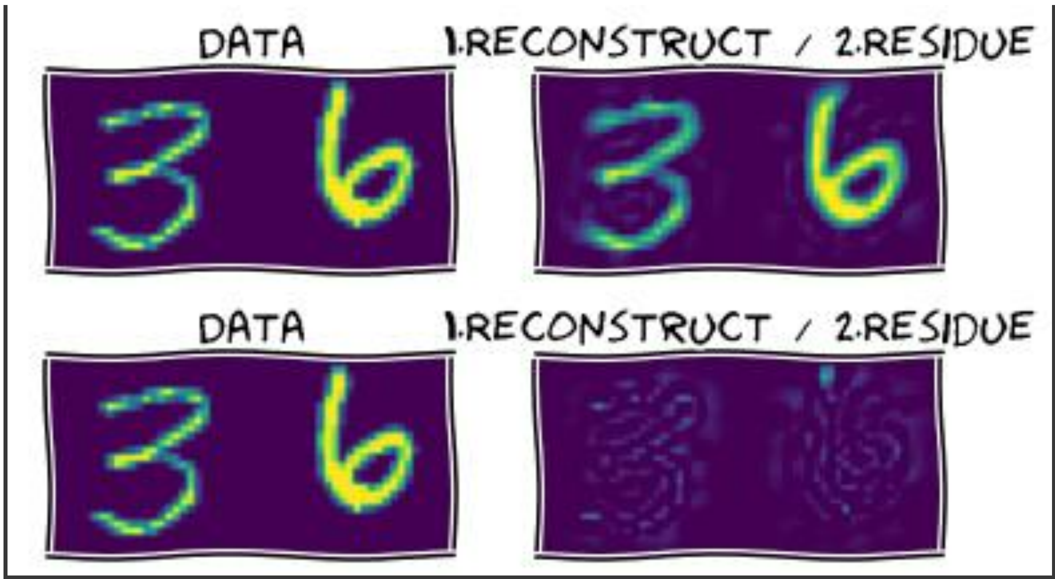
```
(i) ---->  For K=10
```



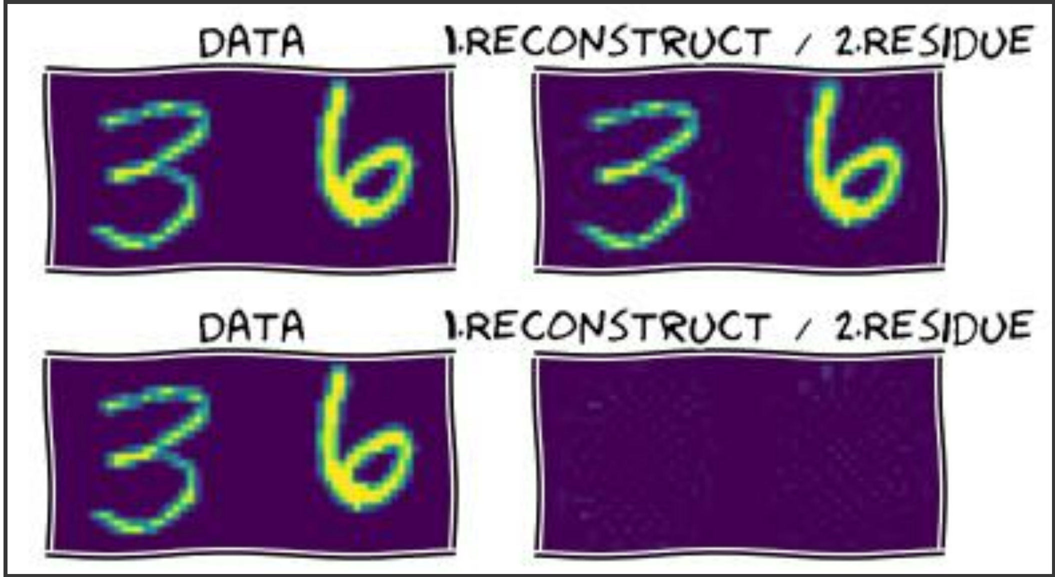
```
(ii) ---->  For K=50
```



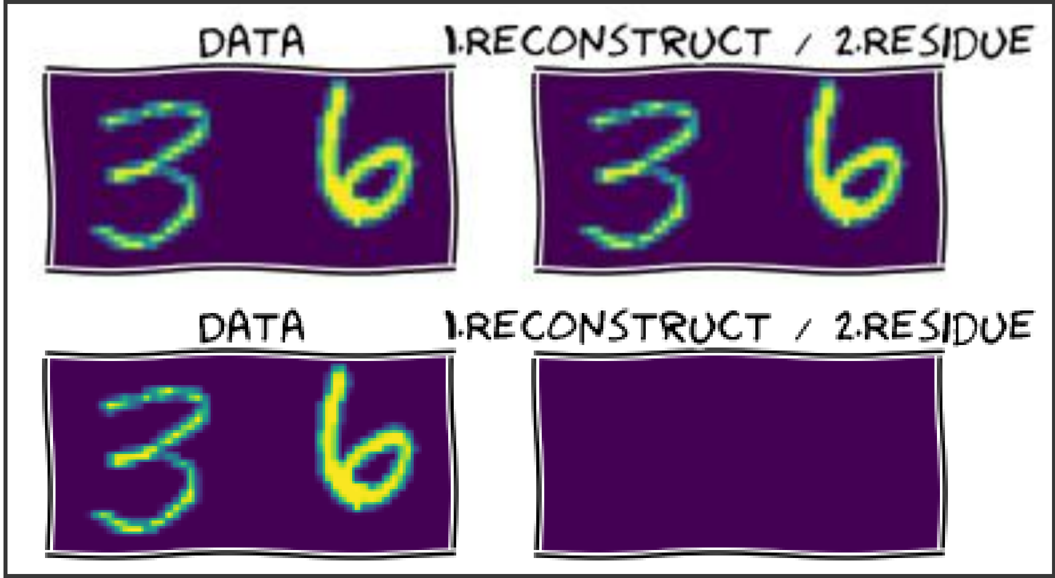
```
(iii) ---->  For K=100
```



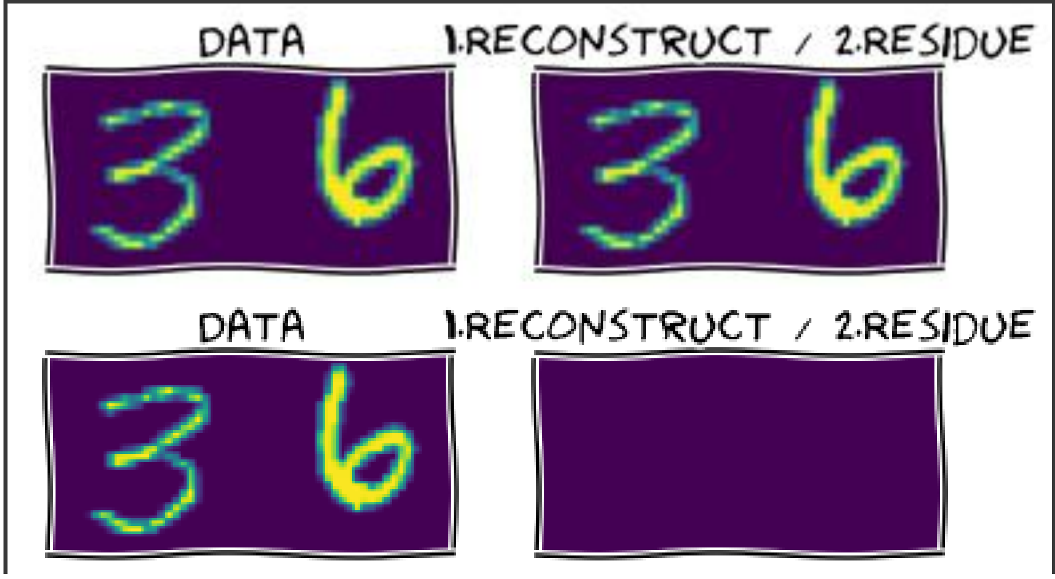
```
(iv) ---->  For K=300
```



```
(v) ---->  For K=700
```



```
(vi) ---->  For K=784
```

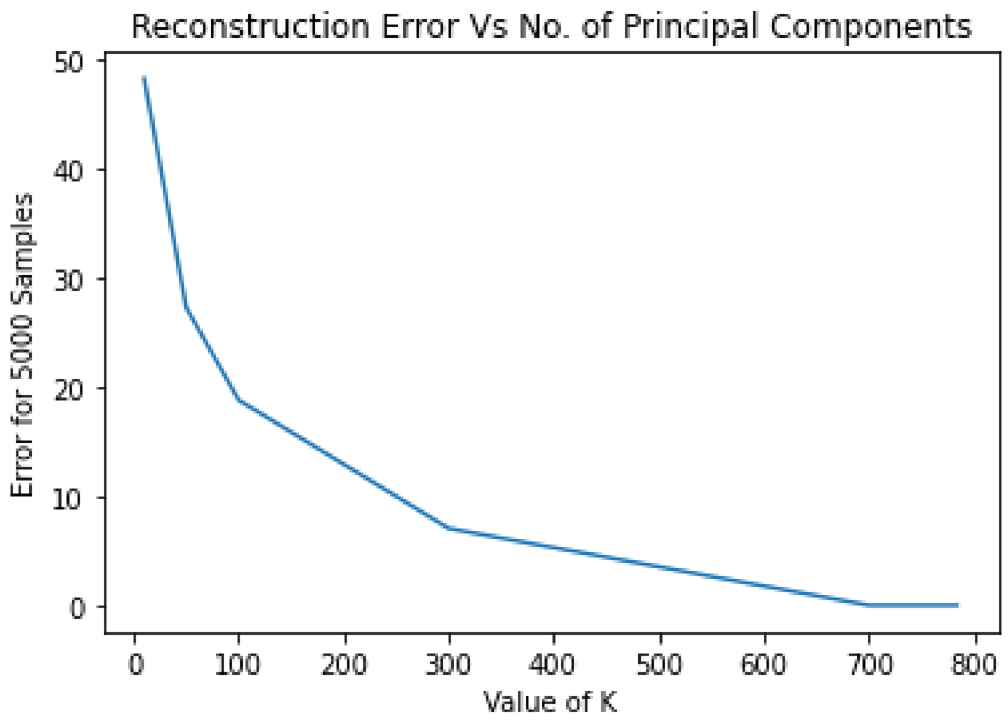


We can see here that as we include more and more principal components we have better reconstruct and less residue which is very well intuitive given that we are using more and more variance, so the quality increases, but what is fascinating is the fact that we can make out the figure easily with only K=100 and so it will save us on space and data, which is why we can also use it to compress images.

2.6) Now we have to plot Reconstruction Error Vs No. of Principal Components, now I have made a triple loop that loops through all 5000 samples and 784 features for K=[10,50,100,300,700,784] and calculates RMS error for all and then I plotted the required graph.

```
k_list=[10,50,100,300,700,784]
error_list=[]

for i in k_list:
  matrix_reconstructed,matrix_residual = reconstruct(reducedMatrix, sor
tedEigenvectors, i)
  sum1=0
  for k in range(5000):
    sum2=0
    for n,m in zip(matrix[k],matrix_reconstructed[k]):
      sum2+=(n-m)**2
    sum1+=np.sqrt(sum2/784)
  error_list.append(sum1/5000)



plt.plot(k_list,error_list)
plt.xlabel('Value of K')
plt.ylabel('Error for 5000 Samples')
plt.title('Reconstruction Error Vs No. of Principal Components')
plt.show()
```

OUTPUT:



We can see that error reduces as we include more principal components for reconstruction.

Resources :
https://compneuro.neuromatch.io/tutorials/W1D5_DimensionalityReduction/student/W1D5_Tutorial3.html#helper-functions

Google Colab link :

https://colab.research.google.com/drive/13pmOi1oraeBQZ3YXzBZNwgQtyRHoP-b_?usp=sharing

---- Abhay Pratap Singh (B20EE084)